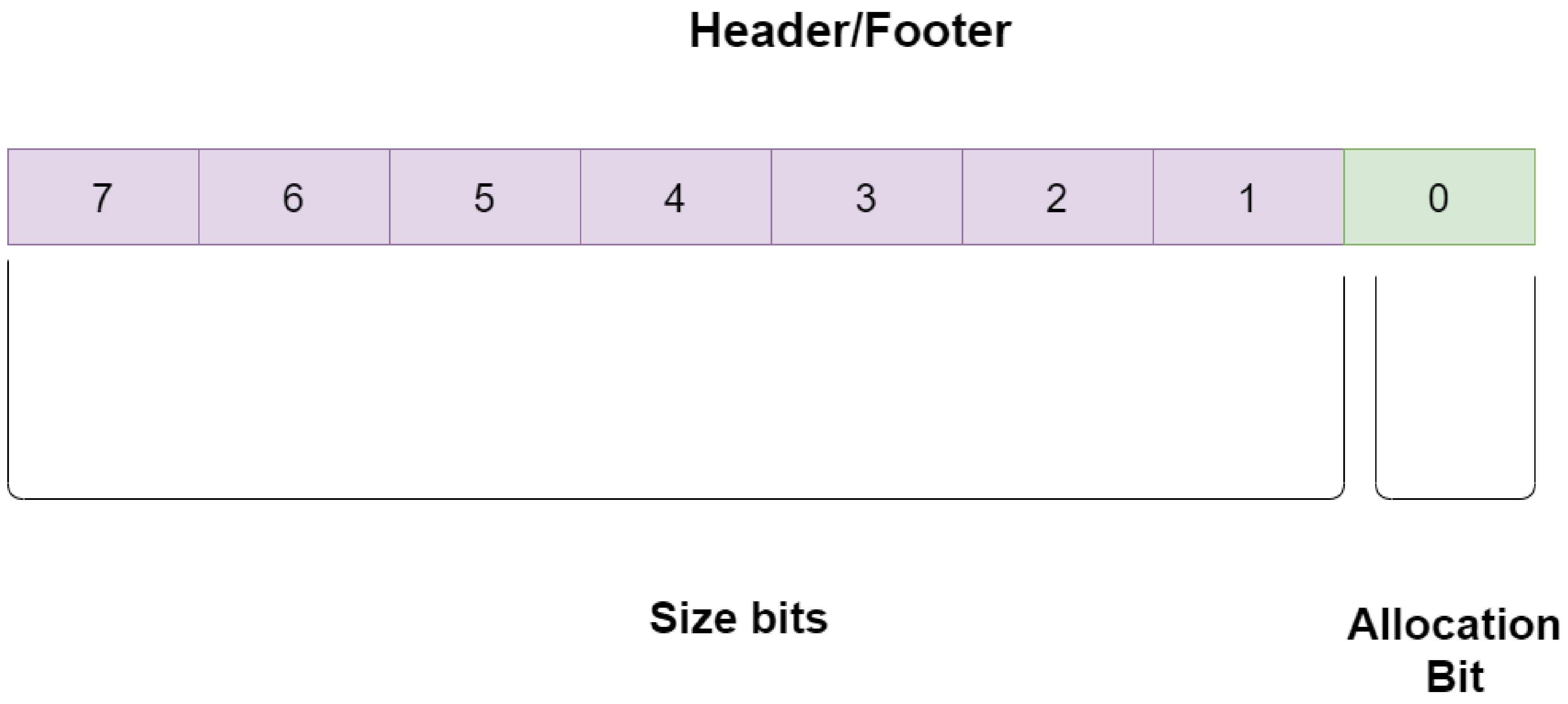
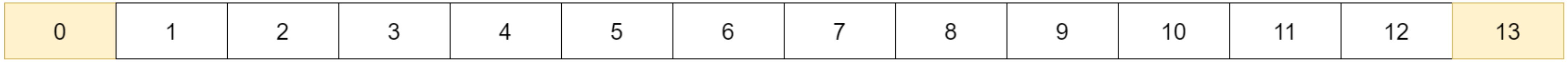
Header/Footer Structure



Sample Execution

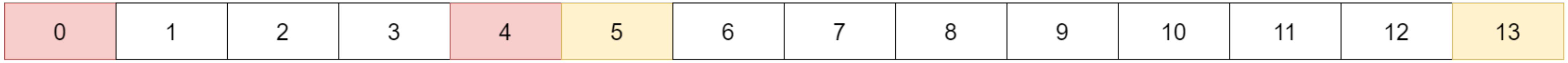
Heap Size = 14

Initial free block



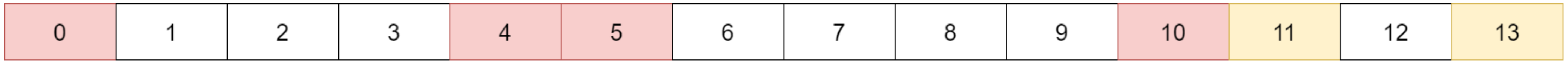
The blocks colored white are empty payload blocks, the blocks in yellow are header and footer blocks.

Command: > **malloc 3**



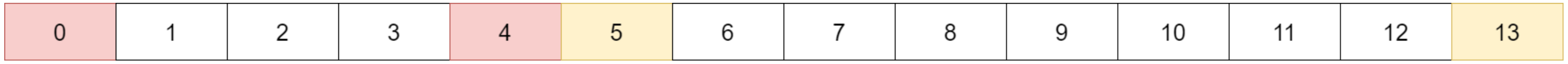
Initially you had a large free block, after the command ‘malloc 3’, A block of size 5 is carved out of the big block. The blocks colored red are allocated block header and footer, and the blocks colored yellow are header/footer of the free block(s).

Command: **> malloc 4**



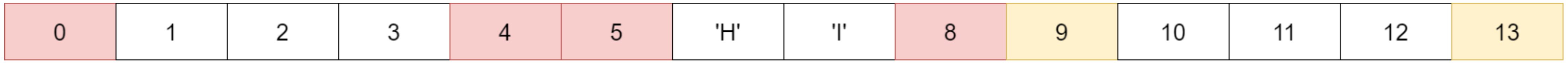
After calling malloc 4, the large free block is again split into chunk of appropriate size. The headers/footers in read are allocated while the ones in yellow are unallocated. Remember, you must always start looking for the free block to allocate from the beginning of the heap. This is the first-fit allocation strategy.

Command: **> free 6**



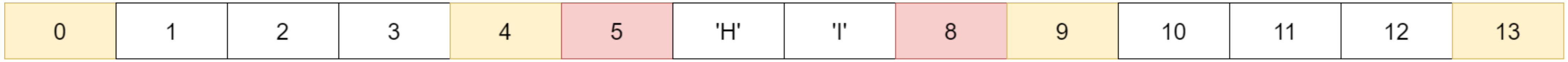
On entering the command ‘free 6’, 6 is the start of the payload of a previously allocated block. The header is located just to the left of it. The header contains the size which you can use to obtain the header of the next block. Your code is required to coalesce free blocks in both forward and backward directions. In this case, the block in question had only a free block after the current block so it merges that free block into itself to create a large free block. Allocation bits and size of newly formed free block are set appropriately.

Command: > malloc 2
> writemem 6 “HI”



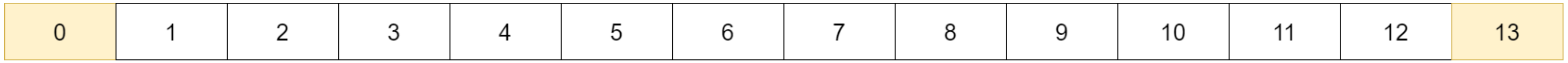
The abovementioned commands will allocate a block of size 2 (excluding header/footer) and the writemem command takes in the index of a block and sequentially writes individual characters from the string into individual blocks. Note that the index need not be a specific start of a payload of a block neither the size of string need to fit into the payload. C doesn’t do bounds checking and such operations are permitted unless it causes violation in which case a Segmentation Violation occurs. Your code also is not required to do bounds checking. If an overwrite does happen, it will destroy the heap structure and that is not supposed to be treated as unfavorable/illegal operation. However, in this case the writemem command doesn’t violate the heap structure.

Command: > **free 1**



On issuing the command ‘free 1’, the block who had their payload starting from 1 is freed. The algorithm looks to the next block to check if it is unallocated so it can merge into a big free block. However, the next block is not free and the block in question is the very first block in the heap. There are 3 distinct blocks now, block 1 of size 5, block 6 of size 4 and block 10 of size 5.

Command: **> free 6**



On entering the command ‘free 6’, the block in question tries both backwards and forwards coalescing. In this case it succeeds in both since there are free blocks just before and just after the allocated block. The block in question can access the header of the next block as well as the footer of the previous block. The footer of the previous block is required for backwards coalescing. All in all, the heap is then again is a giant free block with only one header and footer.